Chapter 17: Building UPS Products

In this chapter we describe the steps you need to take in order to prepare a product for inclusion into the **UPS** framework and then to prepare it for distribution. We go through the steps for a simple case, then discuss the additional steps that may be required in more complex situations. Some sample auxiliary files are provided at the end.

17.1 Basic Steps for Making a UPS Product

In this section we will go through the steps of making a simple, unflavored product compatible with **UPS**. The steps we illustrate in this section are also valid for more complicated situations, but additional steps are generally needed in those cases. These will be noted later in the chapter. We'll use the standard "Hello world" example, with a product **hello**, version v1_0, of flavor NULL. The executable, which is a script in this case, consists of the following text:

```
#!/bin/sh
echo "Hello world"
```

This is a simple case. You don't need any Makefiles or scripts on how to build this product, because it doesn't get built. It runs on all flavors of UNIX without modification, so you should declare it with the flavor NULL. It would be nice to have the `$HELLO_DIR/bin` directory added to your $PATH to use the product, and that's what the setup action will do. The unsetup action will remove `$HELLO_DIR/bin` from your $PATH. No configuration or tailoring is needed, nor are any special actions when the product is declared current.

The steps you need to complete are:

1) Create a directory hierarchy for the product and its related files.

2) Create a `README` file.

3) Create a table file in the location you want it to reside (usually either in the product-specific directory directly underneath your **UPS** development database or in the `ups` directory, if your product has one).

4) Declare the product to your **UPS** development database with the *development* chain so that it doesn't interfere with other peoples' work. Although the product itself doesn't exist yet, the declaration can be done and we recommend it at this stage for convenience.

5) Create the product script in the `bin` directory (or copy it into there).

6) Create man pages (a user's guide is recommended also).

7) Test the product.

17.1.1 Build the Directory Hierarchy

We will take the product root directory to be `hello/v1_0`. This product root directory can sit anywhere in the file system. An appropriate, simple directory structure underneath the product root directory is as follows:

`bin`	contains the executable script `hello`
`man`	contains the unformatted man page(s)
`catman`	contains the formatted man page(s)
`test`	contains the test script(s)

A `README` file should go directly under `hello/v1_0`. We'll put the table file, called `hello.table`, under the database. Remember that most products would have more subdirectories and files than shown here, in particular a `ups` directory as well as `html` and/or `doc` for the user's guide.

17.1.2 Create the Table File

For our example, we'll create the file `hello.table` and put it in the product subdirectory of the development database. A simple table file for this product might look like:

```
FILE=TABLE
PRODUCT=hello
VERSION=v1_0
#
#----------------------
FLAVOR = ANY
QUALIFIERS = ""

ACTION=SETUP
   pathPrepend(PATH, ${UPS_PROD_DIR}/bin, :)
   setupEnv()
```

17.1.3 Declare the Product to your Development UPS

Database

Refer to section 11.1 *Declare an Instance* for instructions on declaring the product to your **UPS** database, or see the reference section 23.5 *ups declare*. In particular, note two things:

1) For an unflavored script like this example, declare the flavor specifically as NULL (using either the **`-f NULL`** or **`-0`** option).

2) Declare it with the chain *development* for your pre-distribution testing (using the **`-d`** option).

For example:

```
% ups declare -0dz /ups_dev_db -r /ups_dev_prod/hello/v1_0 -m\
  hello.table hello v1_0
```

We recommend declaring at this stage for reasons of convenience and organization. It allows you to run **`setup [-d]`** on the product to make the $<PRODUCT>_DIR environment variable available.

17.1.4 Copy the Product Executable to the bin Directory

Create the script in the `bin` directory, or copy or move it to this location.

17.1.5 Provide Product man Pages

See Chapter 39: *Creating and Formatting Man Pages* for more complete instructions on creating man pages.

Create the (unformatted) **nroff** source `$HELLO_DIR/man/hello.1`. It may look similar to this:

```
.TH HELLO 1 LOCAL
.SH NAME
hello - print "Hello world" on stdout
.SH SYNOPSIS
.B hello
.SH DESCRIPTION
.B hello
prints the string "Hello world" on standard output.
```

Use this source to create the formatted man page using the commands[1]:

```
% cd $HELLO_DIR/man
% nroff -man hello.1 > ../catman/hello.1
```

1. If **nroff** is not available, run **`setup groff`** to get the GNU version.

Once it is formatted, it will look like this:

```
    HELLO(1)                                          HELLO(1)


    NAME
    hello - print "Hello world" on stdout


    SYNOPSIS
    hello


    DESCRIPTION
    hello prints the string "Hello world" on standard output.
```

17.1.6 Test the Product

Now you can setup and test your product. As an example, for our product we might run:

```
% setup hello v1_0
% hello
   Hello world
% unsetup hello v1_0
% hello
   sh: hello: command not found
```

In many cases, writing a good test script can be rather challenging. Include at least a basic test to ensure that the product works properly. For our example, the test script just needs to run our **hello** program and verify its output, e.g.,:

```
#!/bin/sh
hello | grep "Hello world" > /dev/null
```

This will exit with a successful exit code if **hello** prints `Hello world`, and fail otherwise.

17.2 Specifics for Different Categories of Products

This section discusses all the steps you need for turning virtually any product into a **UPS** product. We start with the simpler cases and finish with the more complex ones. For all categories of product, if your product has dependencies, either for building or for execution, you need to have them available to you on your development system when you build and test the product.

17.2.1 Unflavored Scripts

Unflavored scripts, that is scripts with the flavor NULL, are the simplest form of **UPS** product. The example in section 17.1 shows how easy it is to create a **UPS** product from an unflavored script. A product like this does not need to be rebuilt on different architectures, and generally does not need CONFIGURE and UNCONFIGURE actions or scripts. Some, although *very few*, unflavored scripts require INSTALLASROOT actions in the table file to copy specific files into `/usr/local/bin`, or to perform similar actions.

We strongly discourage use of `/usr/local/bin` or any other hard-coded path; see section 16.1.1 under 16.1 *Product Development Considerations and Recommendations.*

17.2.2 Pre-built Binaries

Many third-party products obtained from a vendor or downloaded from the Web are binary images without source code. When you go to a vendor's web site, you will often find separate pre-built binaries for several UNIX operating systems/releases. Note that they may use slightly different terminology than we do to refer to the different flavors.

Generally, to run products that consist of executables (as opposed to libraries, for example), you just need to add the executable directories to your $PATH after downloading. To make a product compatible with **UPS**, you should provide a table file that modifies the $PATH, a `README` file and some documentation. If the vendor provides examples and/or any other user files, include them. Most products distributed in this manner include documentation, either man pages or html files, and sometimes both.

Follow this general procedure:

- Create one master product root directory. Underneath it, create the product directory structure, including at least a `bin` directory.

- Create the appropriate product subdirectories (`html` for Web documents, `doc` for PostScript or other forms of documentation, `man` and/or `catman` for unformatted and/or formatted man pages, respectively) and copy the vendor's documentation into them. You can opt to leave the documentation directory structure the same as the way it is provided.
- Create a `README` file in the product root directory with relevant information such as where this product was obtained, by whom, any licensing restrictions or other notes, and so on.
- Create a table file. It can be modified later as needed, but at least a rudimentary table file must exist in its actual location before declaring the product. In most cases, within the table file, the product instance's `bin` directory should be added to the $PATH within a SETUP action, e.g.,:

```
ACTION=SETUP
   pathPrepend(PATH, ${UPS_PROD_DIR}/bin, :)
```

- Create other `ups` directory scripts and data files as needed in the `ups` subdirectory. (For most pre-built binaries you shouldn't need any.)
- Declare the product to a **UPS** database with the chain *development* (**`-d`**) and no flavor (**`-f NULL`**).

Now it's time to create areas for each flavor of the product that you plan to install.

- Duplicate the product root directory tree once for each flavor of binary you plan to install (using **tar** or other appropriate tool).
- For each flavor, copy the pre-built binary into the appropriate `bin` directory. This usually involves unwinding a tar file.
- Declare the suite of product instances (one per flavor) to your **UPS** development database for testing before you distribute them (strongly recommended!).
- Set permissions for all readable files to **`a+r`**. Set permissions for all scripts and other executable files to **`a+x`**.
- Test each one out!

17.2.3 Products Requiring Build (In-House and Third-Party)

Most locally developed products, and many vendor-supplied products, are distributed as source code which must be rebuilt for each OS flavor. We are trying to get away from **UPS**-packaging vendor-supplied products, however, we provide instructions in case you need to do so.

If you are building a product which was obtained from an outside source, you may not have control over the product directory hierarchy. Some outside products include configuration options (via Makefiles) to specify where the resulting libraries and/or images should reside, but in other cases you must give a hard-coded path to the final output file. In the latter cases, when it is absolutely necessary, you may need to use **UPS** as a "bookkeeping" wrapper and common point of distribution. Contact *uas-group@fnal.gov* for assistance.

If you are developing the product yourself, you should follow these guidelines:

- Store the master source code (and all the auxiliary files) in a **CVS** code repository (or other code-version management system) according to your group's policies.
- Use a sensible product directory hierarchy (`src`, `lib`, `bin`, `html`, `doc`, `ups` subdirectories). See section 16.3 for recommendations.
- If the product needs to know its location (or that of its include files or auxiliary files), use the local read-only variable ${UPS_PROD_DIR} or the run-time environment variable $<PRODUCT>_DIR rather than a hard-coded path. Make sure that your table file sets this variable.

Preparation for Rebuilding Any Product

For any product, you first need to create the infrastructure. Much of the work needs to be done only once, and is reused for each flavor of the product that is built:

- Create the master source product directory hierarchy.
- Create/copy `ups` directory scripts, data files, and auxiliary files as needed in the `ups` subdirectory.
- Create at least a basic table file (include QUALIFIERS="BUILD" or "build", and set $<PRODUCT>_DIR under the SETUP action)
- Declare the product with the chain *development* and the flavor `NULL+SOURCE-ONLY` to a local **UPS** database. Make sure that all **UPS** product requirements are declared properly.
- Run **`setup -d -q "?build?BUILD"`** on the product to set $<PRODUCT>_DIR.
- Create source code in the `src` directory, or copy it there.

- Create a Makefile in the product root directory, `${UPS_PROD_DIR}` (or simply write a build script if a Makefile is overkill) to use for building the product binaries. For reproducibility, make sure that you include *all* the steps to go from raw source to the completed product. It is a good idea to have the Makefile or build script run a test suite whenever possible.
- Modify the table file for SETUP and UNSETUP actions.
- Create documentation in the appropriate directories (`html` for Web documents, `doc` for PostScript or other forms of documentation, `man` and/or `catman` for unformatted and/or formatted man pages). Modify table file to note the locations.

  If the documentation came from the vendor in other locations, you don't need to move it; just indicate the locations in the table file.
- Keep track of any relevant information in a `${UPS_PROD_DIR}/README` file. This information should include where the source code came from, any tweaks that were necessary to make it build, the node names and OS versions that were used to build the binaries, known bugs, and so on.
- Set permissions to **`a+x`** for scripts and other executables, and to **`a+r`** for readable files.

Steps for Rebuilding a Product

Once you have created the product structure along with all of the support files, you will need to get down to the business of actually building the product images. If you're planning on redistributing this product to a wider audience than just your machine, you must be careful in selecting a build node. The build nodes should have appropriate levels of compilers, OS, and other products required for building the given product.

We recommend that you create separate build areas, one for each target flavor, so that the different flavors of binary files do not get mixed up. Once you have completed the preparation described above, complete these steps:

- Duplicate this source tree once for each target platform, using the file naming conventions that have been established for your build cluster (use **tar** or other appropriate tool, or you may need to check it out from version control).
- Declare these new directory trees each with its target flavor.

Then for each of the target flavors:

- Declare the product to the database using the flavor, optionally a chain of **`-d`**, and the case-appropriate qualifier *BUILD* or *build* (e.g., **`-q BUILD`**). If this is a product which creates links, make sure they were created properly and that each link points to the *correct* parent product root directory!)

- Setup the product instance of that flavor in order to set $<PRODUCT>_DIR to the right product root directory. Use both the **`-d`** (for development chain, if declared) and **`-q BUILD`** (or **`-q build`**) options (i.e., **`setup -dq BUILD <product>`**).
- Invoke the product's build procedure or Makefile to rebuild the product from scratch.

If this is a product which is building files in a hard-coded path, check to make sure that these files are being created properly. They should reside under the `${UPS_PROD_DIR}` area, but via the symbolic links, they should *appear* to also reside under the hard-coded directory.

17.2.4 Overlaid Products

An overlaid product gets distributed and maintained in the product root directory of its main product. For example, the overlaid products **cern_bin**, **cern_ups**, **cern_lib**, etc., all reside in the product root directory for the main product **cern**. A patch is another good example of the use of overlaid products. The set of products overlaid on a main product is collectively referred to as *the overlay.*

A special keyword, _UPD_OVERLAY, is provided for inclusion in the table file of each overlaid product[1]. _UPD_OVERLAY takes as its value the main product name in double quotes. Its presence indicates that the product is an overlaid product maintained in the root directory of the main product listed as the keyword's value. For example, the table files for the products **cern_bin**, **cern_ups**, and **cern_lib** would contain the following keyword line:

```
_UPD_OVERLAY = "cern"
```

UPD would then use **cern** as the product name when determining the root directory.

In addition to including all the overlaid products as dependencies of the main product, we recommend including the main product as a dependency of each of the overlaid products. This allows separate installation of each of the pieces. Circular dependency lists are allowed in **UPS**.

1. **UPS** regards _UPD_OVERLAY as a user-defined keyword, but it is defined within **UPD**.

17.3 Sample Auxiliary Files

17.3.1 README

Following is the `README` file for the **teledata** v1_0 product. It has been edited for brevity, but shows the kinds of information that are important to include:

```
This is the teledata product.

It contains the HTML files and data files for the Fermilab online
telephone directory.

The files in $TELEDATA_DIR/data are the data files, read by the
teleserver product.  These files are updated daily.

The files in $TELEDATA_DIR/www are the html files, displayed by
the web server.  These files are also updated daily; the A-Z.html
files are rebuilt from raw data, and the index.html, first.html
and last.html are given a new date stamp.

The HTML files must be visible from the web server's default HTML
area.  This is accomplished via links in /usr/local/products
(managed by "ups configure" and "ups declare -c") and links in
the system default HTML directory (handled by the web
administrator).  The /usr/local/products links will be created
automatically when the product is declared.  The web
administrator must create the link in the top-level "default"
HTML directory, via something similar to

     $ cd /path/to/default/html/area
          $ ln -s /usr/local/products/teledata/current/www
telephone

This allows the URL
```

```
http://www-tele.fnal.gov/telephone/

to map to the file

$TELEDATA_DIR/www/index.html
...
The structure of the teledata product is:

$TELEDATA_DIR - parent product directory
  ups - directory containing ups support files
              configure,    unconfigure    -    manage    the
/usr/local/products/teledata links
                current,    uncurrent    -    manage    the
/usr/local/products/teledata/current links
    INSTALL_NOTE - link to this file
  data - directory containing data files
    RAWDATA - raw unprocessed data file
    NASTDATA - processed data file
    email - gdbm index file, keyed on email address
...
For  further  information,  see  the  teleserver  product,  or
please
contact  support person name, telephone and email.
```

17.3.2 INSTALL_NOTE

The following is a sample `INSTALL_NOTE` from the **netscape** v4_5 product:

```
Fermilab installation of Netscape

The Fermilab ups product imposes certain structure upon
its products.  To this end, a wrapper has been provided
which will assist in the downloading and re-structuring
of netscape for use at Fermilab.

To use this tool:

1.  Upd install the install_netscape product.

2.  setup install_netscape.

3.  cd to $INSTALL_NETSCAPE_DIR and execute the
    netscape_install script.  The optional
    argument specifies the directory in which
    to install netscape.  The default is to
```

```
install and declare netscape in
$INSTALL_NETSCAPE_DIR.
```

17.3.3 RELEASE_NOTES

The following is a sample RELEASE_NOTES file from **UPS** v4_3. Notice that for each release of the product, the new update information gets appended to the previous RELEASE_NOTES file contents so as to retain all the update information:

```
UPS v4_3b

Fixed bug in upsact when doing WriteCompileScript for a
product already setup.
EnvSetIfNotSet now has no undo.
Better handling of envremove/pathremove, especially for
cases where the value parameter uses backticks.
Better handling of exeAccess, eliminating the use of 'hash'
in the Korn shell family, and printing error messages as
appropriate.

UPS v4_3a

Fixed problem with ups verify outputting incorrect
information about chains associated
with versions.

UPS v4_3

There are new template files in the ups area for the dbconfig
file and the upsdb_list file.
Many fixes were made to the configuration script,
particularly for NT.
When UPS uses dropit, it will now always use the '-e' switch,
for an exact match.
...
```